Science of Computer Programming 3 (1983) 101–114
North-Holland

EXTENDED NAMING CONVENTIONS FOR COMMUNICATING PROCESSES

Nissim FRANCEZ*

Mathematical Sciences Department, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A.

Communicated by M. Sintzoff
Received April 1983

Abstract. We present two extensions of the *naming conventions* of Communicating Sequential Proceses [11]: *computed communication targets* and *unspecified communication targets*, as well as corresponding extensions to the system of *cooperating proofs* [3] for verifying distributed programs. These language extensions are important for the natural expressibility of many distributed programs. Examples of the use of these extensions are discussed and verified.

1. Introduction

The purpose of this paper is to introduce two extensions of the CSP process *naming conventions* [11] together with their associated proof rules. The suggested extensions permit a more natural expression of many distributed programs than in the CSP original convention. The lack of these features in CSP has been *widely criticized*. The proof rules are presented as an extension of the proof system known as *cooperating proofs*, introduced in [3] and proved to be sound and (relatively) complete in [2]. A similar extension could be incorporated also in the proof system in [12]. The paper assumes acquaintance with the CSP primitives. The proof system of cooperative proofs is briefly reviewed.

A basic aspect of CSP is its *naming conventions*, which require that every communication command have a *target of communication*, i.e., a process that is explicitly named in the command and determinable syntactically at compile time. Thus, for the language as described in [11], a target process must be named either by a simple name, as in *printer*!*line*, or by an indexed name, as in *phil*[$i+1$ *mod* 5]?*leftfork*((appearing in *phil*[i]), in which case the index must be a compile time constant. These conventions have been criticized as being very restrictive for distributed applications in which the establishment of communication

* World-trade Visiting Scientist on sabbatical leave from the Computer Science Department, Technion, Haifa 32000, Israel. This work was partially supported by NSF Grant MCS81-05553 during the author's stay at the Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138. A preliminary version appeared in the 9th ACM-POPL Symposium, Albuquerque, NM, 1982.

is dynamically determined. A similar restriction applies to Milner's CCS [13], where *port-names* are used for specifying the target of communication, instead of CSP's process names. A syntactically determined port-name must be used in any CCS communication command, (see [7]).

The contribution of this paper is the introduction of the following two extensions of the CSP naming conventions:

(1) Computed targets of communication, and

(2) Unspecified targets of communications,

accompanied by appropriate proof-rules which axiomatically define their semantics. The focus on CSP rather than on other languages with similar restrictions is justified by the existence of fully worked-out proof-system for reasoning about CSP programs.

An informal description of these two extensions follows:

(1) *Computed target of communication.* As a typical application where this extension is needed we consider *routing protocols.* Assume that a fixed network structure is given. Also assume that in the solution to some problem a process $p[i]$ has to send a message m to a process $p[j]$, to which it is *not* connected in the network. On the other hand, $p[k]$ is connected to both $p[i]$ and $p[j]$. We would like to be able to express the simple routing strategy whereby $p[i]$ sends a pair (m, j) to $p[k]$ and when $p[k]$ receives the pair, it sends m to $p[j]$. Here $p[k]$ has to communicate with a process whose identity is *dynamically determined* (in the example considered, as a result of a communication). This pattern of communication is not simply expressible in CSP.

As another example, consider programming in CSP a centralized implementation of CSP itself. Such an implementation could consist of a scheduler, receiving information about the processes' readiness to communicate. It will determine the identities of the next pair to proceed by consulting local tables it maintains, and will have to communicate with the chosen pair to let them proceed.

In the CCS context, the need for a similar extension was referred to by [7], where a modeling of UNIX using CCS is suggested. They envision an algebraic, rather than proof-theoretic, definition of an extension to CCS.

(2) *Unspecified target of communication.* For this extension, a typical application is a *service process* (e.g., a process implementing a library procedure or a data-structure). Several critics of CSP (e.g., [5]) have observed that such a process cannot be elegantly coded in CSP. The reason for this inelegance is that the service process has to 'know' the identities of *all* potential users, and explicitly refer to them in its code. Typical code might look as follows:

$$\begin{aligned} server :: *[&user\_1?request(\,) \rightarrow \langle service\text{-}1\rangle \\ &\Box \\ &\vdots \\ &\Box\ user\_n?request(\,) \rightarrow \langle service\text{-}n\rangle]. \end{aligned}$$

Note that whenever a *user* of a *server* is added or deleted, the *server* process must

be reprogrammed. In addition, it is necessary to have many copies of the ⟨*service*⟩ code. We consider an extension in which the target of a communication is left unspecified, and matches a complementary command in *any* process in the network.

We define both extensions by means of extending *cooperating proofs*. The complexity of these extensions may shed some light on the practicality of adding them to any actual implementation of CSP. However, even if one comes to the conclusion that such extensions are too expensive or inefficient, they still could play a useful role in the verification of various protocols expressed as programs in this extended version of CSP. In [14] the verification of a protocol for *remote* servicing is presented which uses these extensions.

We remain within the framework of [11] by considering arrays of processes as the basic structuring tool. However, the results apply also to other structuring methods, i.e., trees of processes. We also retain the constraint that the network structure is fixed, i.e., processes are not dynamically created or destroyed. Relieving the latter restriction calls for more radical modification of the basic language and its proof system, and is considered outside the scope of the current extensions. In a more general setting, one would need operators producing process names as values, e.g., they could be passed as parameters to procedures; however, we shall not consider this generality here, and treat only *computed indices* to process arrays.

We, in fact, show that, within the restrictions imposed, these language extensions can be simulated within the original CSP, and the required properties are deducible from that simulation. One could, of course, argue that such extensions are not needed if they can be simulated. Our argument is that it is better to show such a simulation only once, in a metatheorem, and program using the extensions, thereby avoiding a lot of code copying. A similar justification applies to the usage of **while** or **if-then-else**, even though they can be simulated using *goto*.

The extensions to *cooperating proofs* also serve as a first approximation to proof rules for the ADA [1] rendezvous construct. For such attemtps, see [4, 10], where the [3] system serves as a basis for extension to ADA-restricted versions.

2. Computed communication targets

2.0. The extension

In this section we introduce the extension of CSP allowing for a (dynamically) computed target of communication.

Whenever a process is a member of an array of processes p, we refer to it as a p-process. We use the following syntax for communication commands:

$$p[e]?x, \qquad p[e]!y$$

where e is any expression yielding a value of the index-type of p, and referring only to variables local to the process in which the command appears. The intuitive

meaning of such commands is to input from (or to output to) the process whose index is the *current value* of e. By the disjointness of variable sets of different processes, the value of e may not be changed by another process. We shall assume that the evaluation of e always terminates. (Furthermore, if e's value is outside the boundary of the array of processes, the whole program fails.)

2.1. Partial correctness proofs

To make the paper more self-contained, we briefly review the main features of *cooperating proofs*. The proof system has two levels. In the first level, separate proofs are constructed for each process. To obtain, in the second level, a proof for the concurrent composition, the separate proofs are shown to *cooperate*. The standard proof rules for guarded commands are used in the separate proofs, and in addition, the following *two* i/o axioms:

(A) $\{P\}\alpha\{Q\}$ where α is $p[i]?x$ or $p[j]!y$.

P and Q are *any* assertions referring to the local variables of the process in which the assertions appear. We assume some fixed *first order language with equality* as the assertion language.

The input axiom may be used to 'guess' the input value sent by another process. Note that the postcondition Q cannot really be arbitrary since it must pass a cooperation test. For a more detailed explanation of the output axiom, the reader is referred to [3]. Similar input and output axioms are used also in [12]. As can easily be seen, the soundness of these axioms *does not* depend on the fact that i is a compile time constant. We shall also use the same axioms for computed communication targets.

For the cooperation tests, a *global invariant* I (possibly referring to variables of *all* processes and to auxiliary variables) is introduced. Every process is annotated with *brackets* in such a way that a *bracketed section* contains at most one communication command. The invariant is only required to hold outside bracketed sections.

Since communication targets are syntactically determined, it is possible to use the notion of a *syntactic matching* between a pair of i/o commands in the original formulation of cooperating proofs. The cooperation test has to establish the postconditions of the matching sections and the invariant, given that the precondition and the invariant hold initially. The invariant is also used to rule out all syntactically matching pairs of bracketed sections which *do not* semantically match. The basic cooperation test is:

(C1) $\{pre_1 \wedge pre_2 \wedge I\}\, S_1 \| S_2\, \{post_1 \wedge post_2 \wedge I\}$

where $\{pre_1\}\, S_1\, \{post_1\}$ and $\{pre_2\}\, S_2\, \{post_2\}$ are syntactically matching bracketed sections, and the preconditions and postconditions are taken from the corresponding separate proofs.

In case S_1 and S_2 do not semantically match, the conjunction of the precondition and the invariant will be inconsistent. In [3] additional rules are introduced to reduce (C1) to a sequential proof (not involving concurrency and communication); the most important is the *communication axiom*:

(CA) $\{true\}\, p[j]?x \| q[i]!y\, \{x = y\}$

provided $p[j]?x$ and $q[i]!y$ are taken from $q[i]$ and $p[j]$, respectively. This axiom correctly specifies the effect of such a communication (jointly with the rules governing the preservation axiom, which fix the output command as not changing the value of the output variable; this may have not been emphasized strongly enough in [3]). A substitution also could be used, as this axiom participates only in proof of cooperation, where variables of both matching sections are accessible.

We now extend (C1) to the case of computed communication targets.

Definition. (1) Two communication commands of the form $p[e]?x$ and $q[e']!y$ are *potentially matching* iff they appear in a q-process and p-process, respectively. (Hence, $p[e]?x$ and $p[e']!y$ are always potentially matching.)

(2) Two bracketed sections $\{pre_1\}\, S_1\, \{post_1\}$, $\{pre_2\}\, S_2\, \{post_2\}$ are *potentially matching* if the corresponding communication commands are.

(3) A bracketed section S containing a communication command referring to $p[e]$ is *proper* if S does not affect the value of e. In the sequel, we assume all bracketed sections to be proper.

The notion of *potential matching* replaces that of syntactic matching. A potential match will be an *actual match* provided the following two conditions hold:

(a) The sections S_1 and S_2 are taken from $q[i]$ and $p[j]$, respectively.

(b) There is a time instant during execution when the controls of $q[i]$ and $p[j]$ are about to execute S_1, S_2, respectively, and the current values of e and e' are j and i, respectively. Since S_1, S_2 are proper, the values of e, e' do not change by the time the i/o commands are reached.

As a consequence, we obtain the following extension of the cooperation test (C1):

(DC1) $\{pre_1 \wedge pre_2 \wedge I \wedge e = j \wedge e' = i\}\, S_1 \| S_2\, \{post_1 \wedge post_2 \wedge I\}$

whenever S_1 and S_2 are proper potentially matching bracketed sections taken from $q[i]$ and $p[j]$, respectively. The assertions pre_1, pre_2, $post_1$, $post_2$ are as in (C1).

Note that when e and e' are both compile time determined, potential matching coincides with syntactic matching, and the rule (DC1) reduces to the original rule (C1). Also, in that case all bracketed sections are trivially proper.

The new cooperation test involves *every pair* of potentially matching bracketed sections; hence the proof burden may be heavy. However, in many practical cases the invariant I will be consistent with only one pair of values for e and e', and most of the cooperation tests will vacuously succeed.

2.2. Deadlock freedom proofs

By a *deadlock* we mean a situation where every process in the program has either terminated or is waiting for a communication; hence, all non-terminated processes are *blocked*. This is similar to the deadlock treated by [15]. Sometimes, a stronger notion is discussed where a blocked group of mutually-blocking processes exist, while other processes are executing. If the other processes terminate eventually, then the stronger deadlock reduces to the one defined here. If they do not, the meaning of the whole program is taken to be that of non-termination anyhow. Thus, the stronger notion may fit programs with loosely-coupled processes, which do not intend to terminate, e.g., operating systems (this distinction was not stressed in [3]).

As is well known, an orthogonal approach of *deadlock-detection* (instead of proving deadlock-freedom), is also discussed extensively in the literature, but is not related to the current approach and suggested extensions.

To prove that a program is *deadlock-free*, it is sufficient to *identify* all *blocked situations* (n-tuples of assertions that hold when a program is in a deadlock state; see [3] for a rigorous definition) and show that they are *not* consistent with the invariant I. The CSP convention of *distributed termination* (by which a process may exit a loop if all processes referred to by guards with a true Boolean part have themselves terminated), causes some situations to be unblocked which would otherwise be blocked. To cope with this convention, the propositional variables $END_{q[j]}^{p[i]}$ are introduced. The value of $END_{q[j]}^{p[i]}$ will become true whenever $p[i]$ exits a loop due to this convention and the termination of $q[j]$.

For simplicity of notation, we assume just *one* array p and the corresponding variables END_j^i. Since I may refer to END_j^i, we need an extra cooperation test to establish that I holds after loop exit due to the above convention. Consider a program section (in process $p[i]$) of the form

$$S :: *\Big[\ \Box_{j=1,\ldots,m}\ b_j;\alpha_j \to S_j\Big].$$

Let $A \subseteq \{1,\ldots,m\}$ be any set of guard indices, and let C be the set of all process indices referred to by α_j, $j \in A$. Also, let

$$pre(S,A) \overset{\text{df}}{\equiv} pre(S) \wedge \bigwedge_{j\in A} b_j \wedge \bigwedge_{j\notin A} \neg b_j.$$

Then, we have the following second cooperation test:

$$\text{(C2)} \qquad \Big(\bigwedge_{j\in C} post(p[j]) \wedge pre(S,A) \wedge I\Big) \supset \big(post(S) \wedge I[true|END_j^i]_{j\in C}\big).$$

Here $I[true|END_j^i]_{j\in C}$ is the assertion obtained from I by substituting true for all free occurrences of END_j^i in I corresponding to $j \in C$. Intuitively, this clause states that when $p[i]$ exits the loop (modeled by $post(S)$ becoming true) due to termination

of all processes referred to in guards indexed by A (modeled by $post(p[j], j \in C)$, the invariant is reestablished.

In order to extend this rule to the context of computed targets, let e_j be the computed target of α_j (in S). Let $A \subseteq \{1, \ldots, m\}$ and let $C \subseteq \{1, \ldots n\}$ be such that $|C| \leq |A|$. The roles of A and C are as in (C2). Then, we must show

$$\text{(DC2)} \quad \Big(\bigwedge_{k \in A} e_k \in C \wedge \bigwedge_{j \in C} post(p[j]) \wedge pre(S, A) \wedge I \Big) \supset \big(post(S) \wedge I[true | END_j^i]_{j \in C} \big).$$

The meaning of the rule is as above, but we must show that the values of all the e_j's are indeed in C. Since C is a finite set, say $C = \{i_1, \ldots, i_m\}$, the notation $e_k \in C$ is just a shorthand for $\bigvee_{l=1,\ldots,m} e_k = i_l$.

Next, a theorem stating the condition for *freedom of deadlock* is presented. As mentioned above, in order to show that deadlock is impossible, we must show that no blocked situation is consistent with the global invariant. This is the idea of Theorem 1 in [3]. With dynamic targets of communication, blocked situations can no longer be syntactically determined, since they depend on the run-time values of process indices. Hence, in order for an n-tuple of appropriate assertions to be blocked, it has to satisfy the following condition (*no dynamic matching*):

(NDM) For any pair of assertions q_i and q_j belonging to potentially matching communication commands $p[e]?x$, $p[e']!y$ taken from (proper) bracketed sections S_1 (in $p[j]$) and S_2 (in $p[i]$), respectively, $q_i \wedge q_j \supset (e \neq i \vee e' \neq j)$ holds.

Using this modified definition of a blocked situation, we may now restate Theorem 1 of [3].

Theorem 1. *Given a proof of* $\{P\}[p[1]\| \ldots \|p[n]]\{Q\}$ *with a global invariant* I, p *is deadlock free if, for every blocked situation* $\langle P_1, \ldots, P_n \rangle$, $\sim(\bigwedge_{i=1,n} P_i \wedge I)$ *holds.*

The proof burden is clearly heavier for the case of dynamic targets, since more situations must be considered as (possibly) blocked.

2.3. Soundness and (relative) completeness (for a fixed network)

In order to justify the proof rules for the suggested extension, we show that the *soundness* and (relative) *completeness* are reducible to the corresponding properties for the unextended system, which were proved in [2].

As was shown in [8], it is sufficient to consider i/o commands as guards only. Consider an input command serving as a guard in $p[e]?x \rightarrow S$ (a similar argument applies for output guards). In order to simulate this command, we rely on the fact that the network structure is fixed. Assume that p is declared with index bounds $1 \ldots np$. We replace the input-guarded command by the guarded selection command shown below:

$$\square_{k=1,\ldots,np} \; e = k ; p[k]?x \rightarrow S$$

(note that k is a bound variable). In other words, since e may vary over a finite range only, we replace the computed target by a nondeterministic choice of every element in this range, provided it is the current value of e. If the value of e is not in range, *all* the equalities $e = k$ are false and the whole construct fails. If the original command contains a Boolean part b, it is repeated in every branch in the simulating nondeterministic selection. We omit it to simplify the notation.

In order to prove that the simulation is correct, we bracket the simulating sections in a way we call *naturally induced*. The bracketing of $p[e]?x \to S$ is repeated for each k in the simulating selection.

Consider any (fixed) proof $\{P\}\, S\, \{Q\}$ using an invariant I for S a program using the computable target extension. We prove the following:

Theorem 2. *Any two potentially matching bracketed sections satisfy the cooperation tests* (DC1), (DC2) (*with respect to an invariant I and some bracketing*) *iff the corresponding simulating selections satisfy* (C1), (C2) (*with respect to the same invariant I and the naturally induced bracketing*).

Proof. Let $L_1::\{pre_1\}\, S_1\, \{post_1\}$ and $L_2::\{pre_2\}\, S_2\, \{post_2\}$ be two potentially matching bracketed sections taken from $q[i_0]$ and $p[j_0]$, respectively. Let the corresponding potentially matching communication commands be $p[e]?x \to \bar{S}$ and $q[e']!y \to \bar{T}$, respectively. Also, let

$$L_1'::\{pre_1\}\Big[\underset{k=1,\ldots,np}{\square} e = k\,;\, p[k]?x \to \bar{S}\Big]\{post_1\}$$

and

$$L_2'::\{pre_2\}\Big[\underset{k'=1,\ldots,nq}{\square} e' = k\,;\, q[k']!y \to \bar{T}\Big]\{post_2\}$$

be the corresponding simulating selection.

(a) (DCI) $\to$ (C1). Suppose S_1, S_2 satisfy (DC1). Hence

$$\{pre_1 \wedge pre_2 \wedge I \wedge e = j_0 \wedge e' = i_0\}\, S_1 \| S_2\, \{post_1 \wedge post_2 \wedge I\} \qquad (*)$$

holds.

In the two simulating selections, the corresponding branches with $k = j_0$ and $k' = i_0$ are syntactically matching sections, and (*) reduces to (C1) for that case.

(b) (C1) $\to$ (DC1). The branches for $k = j_0$ and $k' = i_0$ are syntactically matching and satisfy (*) in case (C1) applies, which implies (DC1) applies for S_1, S_2. No other values for k, k' may satisfy (*). Note that if none of the equalities $e = k$ (or $e' = k'$) hold, (C1) will trivially fail for *all* branches in the selection. In this case, e is out of range, and S is indeed semantically invalid.

Next consider an iteration of the form

$$L::\{pre\} * \Big[\underset{j=1,\ldots,m}{\square} b_j;\, \alpha_j \to S_j\Big].$$

After replacing each α_j containing a computed target by its simulating selection (all embedded in the same loop), we get

$$L'::\{pre\} * \Big[\underset{\substack{j=1,\ldots,m;\\ i_j=1,\ldots,np}}{\square} b_j; e_j = i_j; \alpha_{i_j} \to S_j \Big]$$

(where the sequential numbering of the lines is according to a lexicographic order).

Here np is the length of the process array (we again deal with a single array for simplicity), e_j is the computed target index of α_j and α_{i_j} results from α_j by replacing e_j by i_j.

(c) (DC2)→(C2). Suppose that for some A, C (as above) (DC2) is satisfied by L. Define for L' an index set A' including indices for these lines of L' containing the Boolean part $i_j = a$ for $a = e_j$. Then the set of addressed processes (by members of A') is again C, and clearly L' satisfies (C2) with respect to A' and C.

(d) (C2)→(DC2). Follows directly from the observation that if L' satisfies (C2) with some A and C, A contains at most *one* occurrence of each i_j. □

2.4. An example

We next present a simple example which will serve as an abstraction of more complicated routing protocols. In this example, each process $p[i]$, $i = 1, \ldots, n$, sends the value of its local variable a_i to a process A. The process A, in turn, sends (a_i, i) to another process B, and B responds by sending the received value back to $p[i]$. That value is received by $p[i]$ in the local variable b_i. The task is to show that $\bigwedge_{i=1,n} b_i = a_i$ holds upon termination. We give below the annotated text of the program, including all local assertions, and the global invariant. The variables f, g and h are *auxiliary* and are only need for the proof. Note that the second bracketed section in B, containing an i/o command with a nonconstant target, is indeed proper.

The program, with its pre- and post-conditions, is:

$$\{true\}\, RECYCLER::[A \| p[i = 1, \ldots, n] \| B] \Big\{ \bigwedge_{i=1,n} b_i = a_i \Big\}.$$

The individual annotated processes follow.

$$A::\Big\{ \bigwedge_{k=1,n} h[k] = 0 \Big\}$$
$$* \Big[\Big\{ \bigwedge_{k=1,n} h[k] \neq 1 \Big\}$$
$$\underset{k=1,\ldots,n}{\square} \langle p[k]?x \to h[k] := 1; \rangle \{ h[k] = 1 \wedge \bigwedge_{i \neq k} h[i] \neq 1 \}$$
$$\langle B!(x, k); h[k] := 2; \rangle \{ h[k] = 2 \} \Big] \{ true \}$$

$$p[i]::\{g_i = 0\}[\langle A!a_i \to g_i := 1; \rangle \{g_i = 1\}$$
$$\langle B?b_i; g_i := 2 \rangle \{g_i = 2 \wedge b_i = a_i\}]\, \{b_i = a_i\}$$

$$B::\{f=0\}$$
$$*[\{f=0\}\langle A?(y,id)\to f:=1;\rangle\{f=1\}$$
$$\langle p[id]!y;f:=0\rangle\{f=0\}]\{true\}.$$

The invariant is as follows:

$$I \overset{\text{df}}{\equiv} \forall 1\le i\le n[h[i]=1\supset(x=a_i\wedge g_i=1)] \qquad I(1)$$
$$\wedge$$
$$f=1\supset(g_{id}=1\wedge h[id]=2\wedge y=a_{id}) \qquad I(2)$$
$$\wedge$$
$$\forall 1\le i\le n[g_i=0\supset(f=0\vee id\neq i)]. \qquad I(3)$$

The role of the invariant I in this program is to encode the route that message a_i follows, and is typical of such protocols. For example, $h[i]=1$ implies that the communication between $p[i]$ and A has already taken place.

We next present, in some detail, one of the cooperation tests (DC1). The only case that does not reduce to (C1) is the cooperation between the second bracketed sections of B and the second bracketed section in some (potentially matching) p-process, say $p[i_0]$. We must show that (after substituting the appropriate pre- and post-conditions):

$$\{f=1\wedge g_{i_0}=1\wedge id=i_0\wedge I\}$$
$$B?b_{i_0};g_{i_0}:=2\|p[id]!y;f:=0$$
$$\{g_{i_0}=2\wedge b_{i_0}=a_{i_0}\wedge f=0\wedge I\}.$$

We use a justification which can easily be transformed into a fully formal proof using the rule of *formation* and the communication axiom given in [3] and the standard rules for sequential programs.

(a) The post conditions $f=0$ and $g_{i_0}=2$ are established directly by local assignments. To derive $b_{i_0}=a_{i_0}$ we reason as follows: from $id=i_0$, $f=1$ and $I(2)$ in the precondition, we derive $y=a_{i_0}$, and $b_{i_0}=y$ follows from the communication axiom.

(b) Next, we derive the three clauses of the invariant:

$I(1)$. For $i=i_0$, we derive $h[i_0]=2$ in the precondition (using $f=1\wedge I(2)\wedge id=i_0$), and $h[i_0]$ is not affected in the considered program sections; hence, $I(1)$ holds vacuously for $i=i_0$. For $i\neq i_0$, $I(1)$ was true in the precondition, and none of the g_i's (other than g_{i_0}) changed, as well as x did not change, so $I(1)$ holds again.

$I(2)$. Holds vacuously, since $f=0$ was already shown to hold.

$I(3)$. Holds, since $f=0$ was shown.

Note that for all the 'wrong' $p[i]$'s, the precondition of the cooperation test will yield a contradiction.

Next, we show part of the proof that the *RECYCLER* program is deadlock-free. We shall examine a blocked situation in which the bracketed section containing the communication with a computed target is involved.

One such blocked situation is the following: Assume A has terminated, B is waiting to execute $p[id]!y$, $p[id]$ is in its initial state (i.e., waiting to execute $A!a_i$), and for $i \neq id$, $p[i]$ has also terminated. We shall show that this situation is impossible.

The n-tuple of assertion of which such a situation constitutes, contains:

(1) $\bigwedge_{i=1,n} b_i = a_i$ $\quad (post(A))$

(2) $g_{id} = 0$ $\quad (pre(p[id]))$

(3) $b_i = a_i, i \neq id$ $\quad (post(p[i]), i \neq id)$

(4) $f = 1$ $\quad (pre(\langle p[id]!y; f := 0\rangle \text{ in } B)$

The condition (NDM) is satisfied by assumption.

We now consider the conjunction of the invariant I and all the above assertions, and derive a contradiction. The contradiction is immediate, since $f = 1 \wedge I$ implies $g_{id} = 1$ (clause $I(2)$), contradicting assertion (2) above.

Hence, the above described blocked situation cannot occur. Other blocked situations are treated similarly, and the whole program is deadlock-free.

A proof of a more complicated protocol, based on similar ideas, may be found in [14].

3. Unspecified target for communication

This section describes another extension of CSP's process naming convention: the *unspecified target of communication*. The purpose of this extension is to permit communication between one process, say p, and *any* other process that might wish to communicate with p, without forcing p 'to be aware' of the identities of the other processes. As mentioned in the introduction, a typical application for this extension is the *service process* which may communicate with *any* process in the system that needs the particular service it provides. The following notation will be used:

$?x$: Receive a message (of the type of x) from *any* process ready to send one to the process containing the command. Assign the contents of the message to x.

$!y$: Send the value of y to *any* process ready to receive a message (of the type of y) from the process containing the command.

Note that after such a communication has taken place, the process containing the command may *not* 'know' the identity of the process with which it has communicated (compare with [6], where a built-in function provides this information). This is similar to what happens in the ADA *entry call* [1], where the process containing an entry does not 'know' the identity of the caller during the rendezvous. However, information can be passed to the caller by means of the call parameters. In case

the identity is needed for a follow-up communication with the *same target*, it must be included with the message. In [16], this problem is called the 'returned call' problem. Thus, a typical service process will have the form shown below:

$$server :: *[?(request, id) \rightarrow \langle service \rangle (id); id!result].$$

This is the pattern of communication known also as *remote procedure call.*

The (informal) meaning of the *distributed termination convention* in this context (as a natural extension to that of CSP) is as follows: An occurrence of $?x$ or $!y$ in a guard is considered false iff *all* processes containing a complementary command addressing the first process have terminated. We prefer instead the extension of CSP's distributed termination convention suggested in [9] where a loop is terminated once a *predefined* subset or processes addressed in the loop guards have all terminated. Thus, it is not necessary for *all* processes addressed by loop guards to have terminated in order for a loop to terminate. The reason for this generalization is to avoid the situation in which a program containing two (or more) service processes *never* terminates, since each service process will wait for all the others to terminate, including the other service processes.

Potential matching is now defined as follows:

Definition. $?x$ (or $!y$) appearing in a process $p[i]$, *potentially matches any* process having the complementary command addressing a p-process, i.e., $p[e]!u$ $(p[e]?v)$ or $!u$ $(?v)$.

For this notion of potential matching, we get the following extension of the first cooperation test:

(UC1) (a) $\{pre_1 \wedge pre_2 \wedge I \wedge e' = i\}\, S_1 \| S_2\, \{post_1 \wedge post_2 \wedge I\}$ whenever S_1 and S_2 are potentially matching bracketed sections in different processes, S_1 is taken from $p[i]$ and contains $?x$ $(!y)$, and S_2 contains $p[e']!u$ $(p[e']?v)$.

(b) Same as (C1) (the original rule) whenever S_1 contains $?x$ and S_2 contains $!y$.

With this cooperation test, the burden of proof is even heavier than in the case of the computed target, since matching can occur with any process—not just those within the boundaries of a given array of processes.

The second cooperation test (UC2) for loop exit due to distributed termination is clear and therefore omitted. Both tests are justified with the obvious simulation of $?x \rightarrow S$ by

$$\begin{array}{l} [proc_1?x \rightarrow S \\ \quad \square \\ \quad \vdots \\ \quad proc_p?x \rightarrow S \\ ] \end{array}$$

where $proc_1, \ldots, proc_p$ is the list of *all* other processes in the system. This simulation again depends heavily on the assumption of a fixed network.

From a programming language design point of view, the definition given might be too liberal, in that $?x$ and $!y$ will always match, and may cause programming errors of unintended communications. One could impose a restriction that *at most one* partner in a communication will have an unspecified target. We could not find any useful examples which would be inexpressible using the more restricted matching-rule.

4. Conclusion

In this paper, we presented a natural generalization of the process naming conventions in CSP. We allow for the target process of a communication to be either explictly named, but use dynamically determined names, or be left unspecified. These extensions, though being expressible within CSP in the case of a fixed network, allow many distributed algorithms to be more naturally expressed due to a higher flexibility in target determination ability.

The semantics of the extension was given as an extension to *cooperative proofs*, a proof system for partial correctness assertions (and freedom of deadlock) for CSP.

One can derive a straightforward implementation, inspired by the proof rules and the simulation without CSP. The overhead would be the one caused by having to consider all potentially matching i/o commands, basically treating every communication network as if it were a full graph. Thus, the space for tables built by a compiler becomes larger, as well as the polling time typical to several implementations. Obviously, standard data flow techniques (suitably extended to CSP) could be used to restrict the values of expressions, thereby minimizing the size of the process tables. It seems that the area of compiler optimization for concurrent languages still presents several interesting problems.

More work is needed in extending the naming convention to a setup with dynamic process creation and destruction.

References

[1] Reference manual for the ADA programming language (revised draft), U.S. Department of Defense, ACM-Adatec special publication (1982).

[2] K.R. Apt, Formal justification of a proof system for communicating sequential processes, *J. ACM* **30**(2) (1983).

[3] K.R. Apt, N. Francez and W.P. de Roever, A proof system for communicating sequential processes, *ACM-TOPLAS* **2** (3) (1980).

[4] H. Barringer and I. Mearns, Axioms and proof rules for ADA tasks, *IEE Proc., Pt. E (Computers & Digital Techniques)* **129** (2) (1982).

[5] P. Brinch-Nansen, Private communication (1978).

[6] P. Brinch-Hansen, Distributed processes, *Comm. ACM* **21**(11) (1978)

[7] T.D. Doepner, Jr. and A. Giacalone, A formal description of the UNIX operating system, *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (PODC), Montreal, (1983).

[8] Tz. Elrad and N. Francez, A weakest precondition semantics for communicating processes, *Proc. 5th International Conference on Programming*, Torino, Lecture Notes in Computer Science **137** (Springer, Berlin 1982); Also *Theoret. Comput. Sci.* **27** (1983), to appear.

[9] N. Francez, Distributed termination, *ACM-TOPLAS* **2**(1) (1980).

[10] R. Gerth, A sound and complete Hoare axiomatization of the ADA-rendezvous, *Proc. 9th ICALP*, Aarhus, Lecture Notes in Computer Science **140** (Springer, Berlin, 1982).

[11] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21**(8) (1978).

[12] G. Levin and D. Gries, A proof technique for communicating sequential processes, *Acta informat.* **15** (1981).

[13] R. Milner, *A Calculus for Communicating Systems*, Lecture notes in Computer Science **92** (Springer, Berlin, 1980).

[14] C.N. Nikolaou, E.M. Clarke, N. Francez and A. Schumann, A methodology for verifying request processing protocols, *Proc. ACM-SIGCOM 83, Communications, Architectures, Protocols*, Austin, TX (1983).

[15] S.S. Owicki and D. Gries, Verifying properties of parallel programs: an axiomatic approach, *Comm. ACM* **19** (1976).

[16] E.S. Roberts, E.M. Clarke, A. Evans and C.R. Morgan, Task management in ADA: a critical evaluation for real-time multiprocessing, TR-07-80, Aiken Computation Laboratory, Harvard University (1980).